WebView Materialization*

Alexandros Labrinidis†
Department of Computer Science
University of Maryland, College Park
labrinid@cs.umd.edu

Nick Roussopoulos‡
Department of Computer Science
University of Maryland, College Park
nick@cs.umd.edu

Abstract
A WebView is a web page automatically created from base data typically stored in a DBMS. Given the multi-tiered architecture behind database-backed web servers, we have the option of materializing a WebView inside the DBMS, at the web server, or not at all, always computing it on the fly (virtual). Since WebViews must be up to date, materialized WebViews are immediately refreshed with every update on the base data. In this paper we compare the three materialization policies (materialized inside the DBMS, materialized at the web server and virtual) analytically, through a detailed cost model, and quantitatively, through extensive experiments on an implemented system. Our results indicate that materializing at the web server is a more scalable solution and can facilitate an order of magnitude more users than the virtual and materialized inside the DBMS policies, even under high update workloads.

1 Introduction
There is no doubt that the Web has penetrated our lives. From reading the newspaper and shopping online, to searching for the best prices on books or airplane tickets, the Web is increasingly being used as the means to do everyday tasks. One common denominator for most of these activities, is that the web pages we access are generated dynamically, usually due to personalization [BBC† 98]. Personalized web pages, that are created from base data, are one of the many instances of WebViews. In general, we define WebViews as web pages that are automatically generated from base data, which are typically stored in a DBMS.

Similarly to traditional database views, WebViews can be in two forms: virtual or materialized. Virtual WebViews are computed dynamically on-demand, whereas materialized WebViews are precomputed. In the virtual case, the cost to compute the WebView increases the time it takes the web server to service the access request, which we will refer to as the query response time. On the other hand, in the materialized case, every update to base data leads to an update to the WebView, which increases the server load. Having a WebView materialized can potentially give significantly lower query response times, compared to the virtual approach. However, it may also lead to performance degradation, if the update workload is too high.

The decision whether to materialize a WebView or not, is similar to the problem of selecting which views to materialize in a data warehouse [GM95, Gup97, Rou98], known as the view selection problem. There are, however, many substantial differences. First of all, the multi-tiered architecture of typical database-backed web servers raises the question of where to materialize a WebView. Secondly, updates are performed online at web servers, as opposed to data warehouses which are usually off-line during updates. Thirdly, although both problems aim at decreasing query response times, warehouse views are materialized in order to speed up the execution of a few, long analytical (OLAP) queries, whereas WebViews are materialized to avoid repeated execution of many small OLTP-style queries. Finally, the general case of the WebView materialization problem has no constraints, whereas most view selection algorithms impose some resource constraints (e.g. maximum storage or maintenance window limits [KR99]).

In the next section we briefly describe the architecture of typical database-backed web servers, followed by some motivating examples of WebViews.

1.1 Architecture
When only servicing requests for static pages, the web server simply parses user requests, reads the appropriate files from a disk and sends them to the clients that requested them (Figure 1a). Usually, copies of the requested pages are cached in an intermediate node, the proxy, or at the client site in anticipation of future requests on the same pages. By
replicating pages at the proxy or at the client, web caching strives to eliminate unnecessary data transmissions across the network [Mal98].

On the other hand, in order to serve dynamically generated pages (WebViews), the web server has to be interfaced to a relational database (Figure 1b). In this case, after parsing the user requests, the web server sends the corresponding query to the DBMS, often times via a middleware layer, the application server [Gre99]. Then, the query results are sent back to the web server, which formats them in html and transmits the resulting web page to the client that requested it. Since these web pages are generated dynamically, they are usually marked “non-cacheable” and thus cannot be copied at the proxy or at the client.

Existing database-backed web servers, that publish dynamically generated pages, support either virtual or periodically refreshed WebViews, depending on whether users can tolerate stale results or not. For example, at the online auction site eBay (http://www.ebay.com) we have both types of WebViews. The summary pages for each auction category, which contain a list of all the available items together with the highest bid values, are periodically refreshed every few hours. This means that they can easily become out of date. On the other hand, the WebViews for the individual items are virtual, and are always computed on the fly.

Given the multi-tiered architecture of web servers, there are two more WebView materialization options that can guarantee fresh results and have not yet been used: materializing inside the DBMS and materializing at the web server. For the former, we can use the DBMS to also store the query results in the form of materialized views [GM99], whereas for the latter, we can use the web server’s disk to store WebViews as files [LR99]. By materializing inside the DBMS we avoid expensive recomputation, whereas by materializing at the web server, we also eliminate the latency of going to the DBMS every time, which could lead to DBMS overloading [Sin98]. However, in order to guarantee freshness for both cases, the materialized WebViews need to be immediately refreshed with every update on the base data.

1.2 Motivation

There are many examples of WebViews other than personalized web pages. A search at an online bookstore for books by a particular author returns a WebView that is generated dynamically; a query on a cinema server generates a WebView that lists the current playing times for a particular movie; a request for the current sports scores at a newspaper site returns a WebView which is generated on the fly. Except for generating web pages as a result of a specific query, WebViews can also be used to produce multiple versions (views) of the same data. An emerging need in this area is for the ability to support multiple web devices, especially browsers with limited display or bandwidth capabilities, such as cellular phones or networked PDAs.

Although there are a few web servers that support arbitrary queries on their base data, most web applications “publish” a relatively small set of predefined or parameterized WebViews, which are to be generated automatically through DBMS queries. A weather web server, for example, would most probably report current weather information and forecast for an area based on a ZIP code, or a city/state combination. Given that weather web pages can be very popular and that the update rate for weather information is not high, materializing such WebViews would most likely improve performance. In general, WebViews that are a result of arbitrary queries, are not expected to be shared, and hence need not be considered for materialization. This category would include, for example, WebViews that were generated as a result of a query on a search engine. On the other hand, predefined or parameterized WebViews can be popular and thus should be considered for materialization in order to improve the web server’s performance.

Personalized WebViews can also be considered for materialization, if first they are decomposed into a hierarchy of WebViews. Take for example a personalized newspaper. It can have a selection of news categories (only metro, international news), a localized weather forecast and a horoscope page (for Scorpio). Although this particular combination might be unique or unpopular, if we decompose the page into four WebViews, one for metro news, one for international news, one for the weather and one for the horoscope, then these WebViews can be accessed frequently enough to merit materialization.

Stock server example

One motivating example, which we will use throughout the paper, is that of a stock web server. Such a system can have three types of WebViews: summary pages, individual company pages and personalized portfolio pages. Summary pages list companies either by industry group (e.g. consumer goods, financial, transportation, utilities) or by activity (e.g. most active, biggest gainers, biggest losers). Individual company pages have the latest stock price, graphs at various time-scales (from intra-day to multi-year charts) and pointers to news articles about the company. Finally, person-
alized portfolio pages are expected to have a list of the stocks that one owns, along with calculations for their current value and profits/losses, based on the latest stock prices.

The aforementioned WebView s display a wide variety of access and update patterns. For example, the summary pages based on industry groups are typically less update-intensive than the summary pages based on stock activity (e.g., biggest gainers). Even WebView s of the same category can exhibit substantially different access or update characteristics. For example, individual company WebView s are expected to follow the popularity of the company: heavily traded stocks will correspond to WebView s that are accessed frequently and are also update-intensive.

Existing stock web servers typically generate all of their WebView s on the fly, which results in really poor response times at peak hours. Materialization can improve performance dramatically by precomputing popular WebView s and keeping them up to date in the background, instead of repeating their generation with every request. Although the personalized portfolio WebView s are obviously too specific to be considered for materialization, both the WebView s for individual companies and the summary WebView s are candidates for materialization, even under high update rates. The reason for this is that even if, for example, a stock price is updated 10 times a second, it is beneficial to precompute WebView s that are based on it, if they are accessed more often (e.g., 20 times a second).

1.3 Contributions

In this paper we consider the full spectrum of materialization choices for WebView s in a database-backed web server. We compare them analytically using a detailed cost model that accounts for both the inherent parallelism in multitasking systems and also for the fact that updates on the base data are to be done concurrently with the accesses. We have implemented all flavors of WebView materialization on an industrial strength database-backed web server (WebMat) and ran extensive experiments. We then compared the various materialization choices quantitatively. Our results showed that the policy of materializing at the web server scales substantially better than the other two, and that the virtual policy is better than materializing inside the DBMS, except for a very limited number of cases.

The rest of the paper is organized as follows. In the next section we give an overview of related work. Section 3 presents the three materialization policies and compares them analytically. In Section 4 we discuss the results of our experiments, and in the last section we present our conclusions.

2 Related Work

As we mentioned earlier, although the decision whether to materialize a WebView or not, is similar to the view selection problem in data warehouses, there are a few major differences. The most important ones are the multitiered architecture of database-backed web servers, which raises the question of where to materialize, and the need to perform updates at the web server online, as opposed to data warehouses in which updates are usually offline. WebView materialization is also different from the traditional web caching techniques, since it is targeted at dynamically generated pages and guarantees that the WebView is always up to date. Finally, WebView materialization is performed at the web server, whereas web caching is done at the clients or at proxies.

There is some recent work on caching dynamic web data. The Active Cache scheme [CZB98] supports caching of dynamic web objects at proxies. This is done by allowing servers to supply cache applets to be executed on cache hits at the proxies without contacting the server. In [IC97] the authors present the DynamicWeb cache which has the ability to cache dynamic web pages at the server the first time they are created, and in [LISD99] they provide an API which allows application programs to explicitly add, delete and update cached data. Finally, [CID99] describes an algorithm to identify which cached objects are affected by a change to the underlying data. Unfortunately, none of the aforementioned papers deals with the selection problem: identifying which dynamic data to cache and which not to cache.

Although there is a lot of recent literature on building and maintaining web sites [CFP99, AMM98, FFK+B7], there is little work on the performance issues associated with WebView s. [MMM98] provide an algorithm to support client-side materialization of WebView s, and [Sin98, AMR+B7, FLM98] present algorithms to maintain them incrementally. In [LR99], we presented preliminary results that materializing WebView s at the web server is often times better than computing them on the fly. However, we did not consider materialization inside the DBMS, as we do in this paper.

[FLSY99] consider the problem of automatically optimizing the run-time management of declaratively specified web sites. Although they report considerable speedup rates from view materialization, they dismiss it on the grounds of space overhead. We believe that storage overhead is not an issue when it comes to web servers since it refers to disk space and not main memory.

Finally, by materializing WebView s, we allow the web server to scale up well under peak workloads, by serving slightly stale data. This is one way of performing web content adaptation to improve server overload behavior. [AB99] propose to resolve the overload problem by adapting delivered content to load conditions.

3 WebView materialization strategies

The WebMat system is our implementation of a database-backed web server that can support all flavors of WebView materialization. It has three software components: the web
server, the DBMS and the updater. Each of them typically spawns a lot of processes or threads that run in parallel.

The web server services the access requests. Depending on the materialization policy, it may execute a query at the DBMS or read a file from disk. The DBMS computes answers to queries, or applies updates to tables. Finally, the updater runs in the background and services the update stream. It supplies the DBMS with updates to the base tables and may also cause the refresh of derived data inside the DBMS, or write the new version of a WebView to disk, by executing the appropriate query at the DBMS, formatting the results in html, and saving them to a file.

One important property of the WebMat system is transparency: clients sending access requests to the web server do not have to know what kind of materialization a WebView has, if any.

### 3.1 WebView Derivation Path

Before describing the materialization policies in detail, we give an overview of the derivation path for each WebView. First, a set of base tables, the sources, give an overview of the derivation path for each WebView. Before describing the materialization policies in detail, we

Table 1 illustrates how WebView derivation works for the stock server example. In order, for example, to generate the WebView for the biggest losers, we start from the base table with all the stocks (the source), and issue a query to get the ones with the biggest decrease (the view) and, then, format the query results into html (the WebView).

We will use \( s_i \) to denote a source table, and \( S_i = \{ s_{i_1}, s_{i_2}, \ldots, s_{i_m} \} \) for a set of sources. Similarly, we will use \( v_i \) for a view, and \( V_i = \{ v_{i_1}, v_{i_2}, \ldots, v_{i_n} \} \) for a set of views. Finally, we will use \( w_i \) for a WebView, and \( W_i = \{ w_{i_1}, w_{i_2}, \ldots, w_{i_m} \} \) for a set of WebViews.

Formally, if \( S_i \) is the set of sources, we define the query operator \( Q \), such that \( Q(S_i) = v_i \), where \( v_i \) is the view corresponding to the query results. Moreover, we define the formatting operator \( F \), such that \( F(v_i) = w_i \), where \( w_i \) is a WebView, the result of formatting view \( v_i \) into html. If we want to associate a view \( v_i \) with the set of sources that generated it, we use the inverse query operator \( Q^{-1} \): \( v_i = Q^{-1}(S_i) \). Similarly, to associate a WebView with the view it was generated from, we use the inverse formatting operator \( F^{-1} \): \( w_i = F^{-1}(V_i) \). Finally, since there can be a hierarchy of views, we extend \( Q \) to take as argument other views, if necessary. So, for example, in the general case we have \( Q(S_i) = v_1^1 \), \( Q(v_2^1) = v_2^2 \), \ldots, \( Q(v_i^{n-1}) = v_i^n \), \( w_i = F(v_i^n) \). If \( n = 1 \), we have a flat schema.

All WebViews have the same derivation path regardless of the materialization policy. The only difference among the three policies is that the materialized strategies choose to cache (and keep consistent) parts of the intermediate results, whereas in the virtual strategy everything is computed from scratch. In the next sections, we describe the three policies in detail.

### 3.2 Virtual Policy

In the virtual policy, everything is computed on the fly. To produce a WebView \( w_i \) we would need to query the DBMS and format the results in html. Therefore, the cost of accessing WebView \( w_i \) would be:

\[
A_{virt}(w_i) = C_{query}(S_i) + C_{format}(v_i) \tag{1}
\]

where \( v_i = F^{-1}(w_i) \) is the view from which the WebView \( w_i \) is generated, \( S_i = Q^{-1}(v_i) \) is the set of sources needed to answer the query, \( C_{query}(S_i) \) is the cost to query the sources, and, \( C_{format}(v_i) \) is the cost of formatting view \( v_i \) into html. We notice that the query part of the access cost is executed at the DBMS, whereas the formatting part is performed at the web server.
Since nothing is being cached under the virtual policy, whenever there is an update on the base tables that produce the WebView, we only need to update the base tables. Therefore, the cost of an update to source \( s_j \) is:
\[
U_{\text{virt}}(s_j) = C_{\text{update}}(s_j) \quad @ \text{dbms}
\]  
(2)
where \( s_j \) is one of the base tables that are used to produce WebView \( w_i \), or \( s_j \in S^{-1}(F^{-1}(w_i)) \), and \( C_{\text{update}}(s_j) \) is the cost to update table \( s_j \).

We realize that the formatting of the query results during accesses can be done in parallel with the updating of the sources, as they are done at different processes (the former is being done at the web server, while the latter is done at the DBMS). However, we also realize that there is a possible source of data contention between the query phase during the accesses and the updates, since they both have to be done at the DBMS.

### 3.3 Materializing inside the DBMS

When materializing inside the DBMS (the matdb policy) we save the results of the query that is used to generate the WebView. To produce the WebView, we would need to access the stored results and format them into html. Therefore, the access cost for WebView \( w_i \) in this case is:
\[
A_{\text{matdb}}(w_i) = C_{\text{access}}(v_i) + C_{\text{format}}(v_i) \quad @ \text{dbms} \quad @ \text{web server}
\]  
(3)
where \( v_i = F^{-1}(w_i) \) is the view from which the WebView \( w_i \) is generated, and \( C_{\text{access}}(v_i) \) is the cost of accessing the materialized view \( v_i \). We notice that, similarly to the virtual policy, the first part of the access cost is executed at the DBMS, whereas the formatting part is performed at the web server.

Since we assumed a no staleness requirement, the stored query results need to be kept up to date all the time. This means that on every update to one of the base tables \( s_j \) that produce the WebView, we have to regenerate the WebView from scratch and save it as a file for the web server to read. Therefore, the cost of an update to source \( s_j \) is:
\[
U_{\text{matdb}}(s_j) = C_{\text{update}}(s_j) + \sum_{v_k \in V_j} C_{\text{update}}(v_k) \quad @ \text{dbms}
\]  
(4)
where \( C_{\text{update}}(s_j) \) is the cost to update source \( s_j \), \( V_j \) is the set of materialized views that are affected by the update to table \( s_j \), or \( V_j = \{v_m | s_j \in S^{-1}(v_m)\} \), and, \( C_{\text{update}}(v_k) \) is the cost to update the materialized view \( v_k \). There are two options for updating the materialized view \( v_k \): incremental refresh and recomputation. For the incremental refresh case, the cost to update \( v_k \) is simply:
\[
C_{\text{update}}(v_k) = C_{\text{refresh}}(v_k) \quad @ \text{dbms}
\]  
(5)
whereas in the recomputation case, the cost to update \( v_k \) is:
\[
C_{\text{update}}(v_k) = C_{\text{query}}(S_k) + C_{\text{store}}(v_k)
\]  
(6)
where \( S_k = Q^{-1}(v_k) \) is the set of sources needed to answer the query that corresponds to view \( v_k \), and \( C_{\text{store}}(v_k) \) is the cost to store the query results inside the DBMS, which includes the cost to delete the previous “version” of \( v_k \). Although the incremental refresh is expected to have the lowest cost, there are classes of views which cannot be updated incrementally and thus must be recomputed every time.

We realize that, like the virtual case, the formatting of the query results during accesses can be done in parallel with the updating of the sources and the materialized views, as they are done at different processes (the former is being done at the web server, while the latter is done at the DBMS). However, there is a possible source of data contention between the queries executed during the servicing of the access requests and the updates of the sources or of the materialized views, since they are all done at the DBMS.

### 3.4 Materializing at the web server

When we materialize a WebView at the web server (the matweb policy) we do not need to query the DBMS or perform any further formatting in order to satisfy user requests. We simply have to read it from disk, which makes the cost of accessing a WebView \( w_i \) rather small:
\[
A_{\text{matweb}}(w_i) = C_{\text{read}}(w_i) \quad @ \text{web server}
\]  
(7)
where \( C_{\text{read}}(w_i) \) is the cost to read \( w_i \) which has been saved as a file to disk.

Because of the no staleness requirement, the materialized WebView needs to be kept up to date all the time. This means that on every update to one of the base tables \( s_j \) that produce the WebView, we have to recompute the WebView from scratch and save it as a file for the web server to read. So the cost of an update to source \( s_j \) is:
\[
U_{\text{matweb}}(s_j) = C_{\text{update}}(s_j) + \sum_{v_k \in V_j} C_{\text{query}}(S_k) \quad @ \text{dbms}
\]
\[+ \sum_{v_k \in V_j} C_{\text{format}}(v_k) + C_{\text{write}}(w_k) \quad @ \text{updater}
\]  
(8)
where \( V_j \) is the set of views that are affected by the update to table \( s_j \), or \( V_j = \{v_m | s_j \in S^{-1}(v_m)\} \), \( v_k = F^{-1}(w_k) \) is the view that generates WebView \( w_k \), \( S_k = Q^{-1}(v_k) \) is the set of sources needed to answer the query that corresponds to view \( v_k \), and \( C_{\text{write}}(w_k) \) is the cost to write the WebView \( w_k \) to disk.

We realize that the handling of user requests and the updates can be done entirely in parallel. Moreover, parts of the execution of an update can also be done in parallel, since the work is distributed among the DBMS and the updater.
The access costs than the update costs.

The problem is to minimize the background while the server is processing access requests.

Unlike traditional materialized view applications, updates in taking into account the frequencies with which they occur.

We will need to aggregate the access and update costs, for the cost functions presented in Sections 3.2 - 3.4.

In order to solve the WebView selection problem, except that have to be queried to generate the WebViews in \( W_{\text{virt}} \), or \( S_{\text{virt}} = Q^{-1}(F^{-1}(W_{\text{virt}})) \), and similarly \( S_{\text{matdb}} \) the set of sources that have to be queried to generate the WebViews in \( W_{\text{matdb}} \), and \( S_{\text{matweb}} \) the set of sources needed for generating \( W_{\text{matweb}} \).

Since we are minimizing the average query response time, in order to calculate the total cost we simply need to identify for each policy how much the concurrent updates influence the access requests. Table 2 lists which subsystems are involved when servicing (a) access or (b) update operations under each policy. For example, when a WebView is accessed under the \( \text{virt} \) policy, both the web server and the DBMS are involved (Table 2a, first line). The same holds for the \( \text{matdb} \) policy (second line), whereas for accessing WebViews under the \( \text{matweb} \) policy only the web server is required (third line). On the other hand, the DBMS is required for servicing updates under all three policies (Table 2b), whereas the updater processes are involved only under the \( \text{matweb} \) policy (Table 2b, third line). We clearly see that the DBMS is used at all times, except for when accessing a WebView which is materialized at the web server. This means that the database server can become the bottleneck of the system, and thus the load on the DBMS is expected to dominate the average query response time.

Let \( TC \) be the total cost for servicing access requests, which is the amount that we want to minimize. Obviously, \( TC \) will include the access costs for the WebViews in our system, but it must also include the influence to the access costs from the updates that are executed concurrently. As we mentioned earlier, the DBMS is expected to be the bottleneck of the system, so we isolate from the update costs the parts that are executed in the DBMS. Formally, we use \( \pi_{\text{dbms}}(C) \) to select from cost \( C \) the part that is executed in the DBMS. From Eq. 2, we have that \( \pi_{\text{dbms}}(U_{\text{virt}}) = U_{\text{virt}} \), and from Eq. 4, \( \pi_{\text{dbms}}(U_{\text{matdb}}) = U_{\text{matdb}} \). To get \( \pi_{\text{dbms}}(U_{\text{matweb}}) \) from Eq. 8 we simply ignore the parts that are executed in the updater processes (third term).

---

### Table 2: Work distribution among processes for each policy

<table>
<thead>
<tr>
<th></th>
<th>web server</th>
<th>DBMS</th>
<th>updater</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{virt} )</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{matdb} )</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{matweb} )</td>
<td>√</td>
<td></td>
<td>√</td>
</tr>
</tbody>
</table>

(a) Accesses

<table>
<thead>
<tr>
<th></th>
<th>web server</th>
<th>DBMS</th>
<th>updater</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{virt} )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{matdb} )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{matweb} )</td>
<td></td>
<td></td>
<td>√</td>
</tr>
</tbody>
</table>

(b) Updates
Putting it all together, we have that the total cost for servicing access requests is:

\[
TC = \sum_{w_h \in W_{\text{mat\-web}}} f_u(w_h) \times A_{\text{mat\-web}}(w_h) \\
+ \sum_{s_k \in S_{\text{mat\-web}}} b \times f_u(s_k) \times \pi_{\text{dbms}}(U_{\text{mat\-web}}(s_k)) \\
+ \sum_{w_i \in W_{\text{virt}}} f_u(w_i) \times A_{\text{virt}}(w_i) \\
+ \sum_{s_i \in S_{\text{virt}}} f_u(s_i) \times U_{\text{virt}}(s_i) \\
+ \sum_{w_j \in W_{\text{mat\-db}}} f_u(w_j) \times A_{\text{mat\-db}}(w_j) \\
+ \sum_{s_j \in S_{\text{mat\-db}}} f_u(s_j) \times U_{\text{mat\-db}}(s_j)
\]

where \( b = 0 \), if \( W_{\text{virt}} = W_{\text{mat\-db}} = \emptyset \), and \( b = 1 \), otherwise. The meaning of \( b \) is that when we only have WebViews materialized at the web server, the cost of updating them in the background using the DBMS does not have a direct impact on the average query response time. However, when we have WebViews that are either virtual or materialized inside the DBMS, the cost of updating the mat\-db WebViews in the background will influence the average query response time of the virt and mat\-db WebViews.

### 3.7 Staleness calculation

Although, at first sight, the virtual policy would seem to provide the most up to date responses, this misconception is quickly cleared away if we consider the basis of our freshness measurement to be the time of the reply instead of the request. Using the time of the reply is more meaningful, since that is the time when the users get to access the answer to their query. We call minimum staleness, \( MS \), the time it takes for an update to propagate to the user, or, in other words, the time between the reply to a WebView request and the time of the last database update that affected this reply. All points of time refer to the web server in order to avoid network delays, so the time of the reply is actually the time the web server sends the reply back to the user and not the time the user receives the reply.

Figure 2(a) illustrates the minimum staleness under the virtual policy (\( \text{virt} \)), which is

\[
MS_{\text{virt}} = T_{\text{update}}(s_j) + T_{\text{query}}(S_i) + T_{\text{format}}(v_i)
\]

before request during request

For the materialized inside the DBMS policy (\( \text{mat\-db} \)), Figure 2(b) gives us

\[
MS_{\text{mat\-db}} = T_{\text{update}}(s_j) + T_{\text{refresh}}(v_i) + T_{\text{access}}(v_i) + T_{\text{format}}(v_i)
\]

before request during request

Finally, Figure 2(c), illustrates that the minimum staleness when materializing a WebView at the web server (\( \text{mat\-web} \)) is

\[
MS_{\text{mat\-web}} = T_{\text{update}}(s_j) + T_{\text{query}}(S_i) + T_{\text{format}}(v_i)
\]

before request during request

By comparing the three minimum staleness formulas, we have:

\[
MS_{\text{mat\-db}} - MS_{\text{virt}} = T_{\text{refresh}}(v_i) + T_{\text{access}}(v_i) - T_{\text{query}}(S_i)
\]

\[
MS_{\text{mat\-web}} - MS_{\text{virt}} = T_{\text{write}}(w_i) + T_{\text{read}}(w_i)
\]

Under light load conditions, we expect to have the virtual policy to have slightly lower minimum staleness than the other two policies: \( MS_{\text{virt}} \leq MS_{\text{mat\-web}} \leq MS_{\text{mat\-db}} \). However, this will not hold when the load at the server increases. As we will see later in the experiments section, all policies do not scale up in the same way. Specifically, the mat\-web policy can support at least 10 times more requests than the other two policies (\( \text{virt, mat\-db} \)), since it allows for more parallelism between the access and update requests. This means that as the load at the system increases, the virt and mat\-db policies will reach the heavy load mark much faster than the mat\-web policy. After that point, the time...
to service access requests increases dramatically and affects the minimum staleness (Figure 3). In general, although under light server loads, the minimum staleness is about the same for all policies, as the load increases in the server, the mat-web policy is expected to have the least minimum staleness, since it scales better.

Figure 3: Minimum staleness under heavy loads

3.8 Discussion
As mentioned during the presentation of the materialization strategies, there is a lot of parallelism in a database-backed web server. For example, the formatting of the query results at the web server can be done in parallel with the updates at the DBMS. In a single-processor machine, this parallelism means that we are able to recover idle time due to I/O blocking or data contention by performing other useful tasks.

Furthermore, we expect that the virtual and the materialized inside the DBMS policies make the database server the bottleneck, since every request (accesses and updates alike) has to query the DBMS. For accesses, this means that each user request has to go through an extra layer of software, communicating data back and forth. On the other hand, the materialized at the web server policy breaks this bottleneck, by performing a lot of the work in the background (the updater processes) and relying on the web server alone to service user requests. This was verified by our experiments, which we present in the next section.

4 Experiments
As we mentioned earlier, the WebMat system consists of three software components: the web server, the DBMS and the updater. We used the Apache\(^2\) web server, version 1.3.6 and the Informix Dynamic Server with Universal Data Option ver. 9.14. The updater was written in Perl.

Web server extensions In order for the web server to generate pages dynamically, we need to execute scripts that communicate with the DBMS. To avoid the overhead of creating a new Unix process with every access request (which is what happens with cgi-bin), we used the mod_perl package version 1.19 on top of the Apache web server. This way, the handling of the WebView access requests was done exclusively from within the apache processes, resulting in an order of magnitude improvement in performance [LR00a]. We used perl DBI (version 1.08) and the Informix DBD (version 0.60) to communicate to the DBMS, from within Apache, as well as from the updater processes. We kept the connections to the database persistent, so that we did not have to establish a new connection with every request, which gave us another order of magnitude improvement in performance. Finally, we also instrumented Apache to measure the time it takes for the server to service each query request. Note that we made our measurements of query response time at the server, thus eliminating any network latency.

Updater We had 10 updater processes running in the background. Informix does not have native support for materialized views, so for the mat-db policy, we stored the materialized views as tables, and had the updater issue an update SQL statement whenever there was an update on the base data. It should be noted that most DBMS products that support materialized views, also store them as relational tables (e.g. Oracle [BD+98]).

Hardware We used a SUN UltraSparc-5 with 320MB of memory, a 3.6GB Seagate Medalist disk as our server, and, a cluster of 22 SUN Ultra-1 workstations as clients. All of the machines were on the same local area network and were running Solaris 2.6.

Workload Unless noted otherwise, in each experiment we had 1000 WebViews that were defined over 10 source tables (100 per table). The queries corresponding to the WebViews were selections on an indexed attribute, which returned 10 tuples each. The WebView size in html was 3KB. Each experiment was executed for 10 minutes. Finally, the update operations were changing the value of one attribute at the source table.

4.1 Scaling up the access rate
In this group of experiments we increased the access request rate from 10 requests per second up to 100 requests per second and measured the average query response time under the three different materialization policies: virtual (virt), materialized inside the DBMS (mat-db) and materialized at the web server (mat-web).

A load of 10 access requests per second should correspond to a “moderate” load at the server of about 0.8 million hits per day. On the other hand, 100 requests per second should correspond to a rather “heavy” load at the web server of about 8.6 million hits per day. For comparison, our department’s web server (http://www.cs.umd.edu) gets about 95,000 requests per day or 1.1 request per second, whereas the widely popular online auction site eBay

---

2 Apache is the most popular web server according to the February 2000 Netcraft Web Server Survey, with a 58% market share. The survey is available online at [http://www.netcraft.com/survey/](http://www.netcraft.com/survey/).
products with native support for materialized views, if they use relational tables to store the materialized views. However, when we also have updates (Figure 4b), except for updating the source tables, the matdb policy has to refresh the materialized views as well. This means that the DBMS (which is the bottleneck) will become significantly more loaded, which results in a substantial drop in performance for the matdb policy, compared to the virt policy. For example at 25 requests/sec, although with no updates the matdb policy is 9.69% faster than the virt policy, when we have 5 updates/sec, the virt policy is 63.53% faster than the matdb policy.

4.2 Scaling up the update rate

In this group of experiments we increased the update rate up to 25 updates/sec, while the access rate was constant at 25 requests/sec. Each experiment was scheduled to run for 10 minutes and was repeated three times, one for each policy (virt, matdb and matweb). We report the average query response times per WebView in Figure 5.

Our first observation is that the average query response time remains practically unchanged for the matweb policy despite the updates. The reason behind this is that, as predicted by the total cost formula of Eq. 9, the cost of the

---

3Of course, eBay does not have just one plain SUN UltraSparc-5 to serve all these hits, but, rather, they rely on many machines. A simple search on the ebay.com domain, lists 478 machines, out of which 35 have the word “cgi” as part of their name and are most probably used to serve dynamically generated web pages.
4.3 Scaling up the number of WebViews

In this group of experiments we varied the number of WebViews in the system. We ran one set of experiments with 100 WebViews, a second set with 1000 WebViews and a third set with 2000 WebViews. In all experiments, the aggregate access rate was 25 requests/sec. Each experiment ran for 20 minutes and was repeated three times, one for each policy (virt, mat-db and mat-web). In all experiments, we modified the view definition for 10% of the WebViews: instead of a simple selection, they were defined as a join on the index attribute between two tables, resulting in a more expensive generation query.

Figure 6a depicts the results of our experiments with no updates and Figure 6b when we have 5 updates/sec. In the no update case, when the number of WebViews is small, the virt policy performs substantially worse than the mat-db policy (3.5 times worse for 100 WebViews, and 21% worse for 1000 WebViews), since the time to compute the WebView generation query is not negligible. However, as the number of views increases, so does data contention. The mat-db policy will exhibit more data contention than the virt policy, because the number of materialized views is much higher than the number of source tables. Eventually (when the number of WebViews is 2000), the performance of the virt policy will be better than that of the mat-db policy, even for expensive queries. If we consider the case with 5 updates/sec, the crossover point where the virt policy outperforms the mat-db policy is even earlier, at 1000 WebViews, whereas for 2000 WebViews, the virt policy gives 43% faster query response times than the mat-db policy.

4.4 Scaling up the WebView size

The size of a WebView can increase in two ways: (a) by increasing the number of tuples in each view, or (b) by increasing the size of the resulting html page. We investigated both options in this group of experiments.

In the first set of experiments, we increased the number of tuples in a WebView from 10 to 20. The access rate was 25 requests/sec, and we also had 5 updates/sec. The experiment run for 10 minutes, and was repeated 3 times, one for each policy. We report the average query response time per WebView in Figure 7. We can see that although the response time increases for the virt policy, it does not double: there is a 50% increase for the virt policy and a 15% increase for the mat-db policy. Moreover, the response time for the mat-web policy remains virtually unaffected, since all the “extra work” generated from the increase in the view size is executed at the updater process and does not have a direct effect on the web server.
In the second set of experiments, we increased the size of the html page (WebView) from 3KB to 30KB. The access rate was 25 requests/sec, and we also had 5 updates/sec. The experiment run for 10 minutes, and was repeated 3 times, one for each policy. We report the average query response time per WebView in Figure 8. Again we see that the response times for the virt and matdb policies increase. However, unlike the previous experiment, in this case, the response time for the matweb policy increases significantly. This is explained by the fact that a big change in the WebView size (from 3KB to 30KB) is actually affecting the web server, since it will have to spend more time reading the files from disk.

4.5 Zipf vs uniform access distribution
In all of our experiments, we used a uniform distribution for the access rates. We ran two sets of experiments where the access rates followed a Zipf distribution with a theta of 0.7 as suggested in [BCF+99] and compared them against the uniform distribution case (due to lack of space we do not include the graphs, the reader is referred to [LR00b]). We saw that the query response times are significantly lower (11% - 23%) under the Zipf distribution for all policies. This is due to the fact that there is more reference locality in the Zipf workload than in the uniform case. Therefore, by using a uniform distribution in our experiments, we exposed the WebMat system to a “worst case” scenario for the access requests.

4.6 Verifying the cost model
In the final set of experiments we tried to verify the total cost formula from Eq. 9. We had 1000 WebViews (500 of them were kept virtual and 500 were materialized under the matweb policy), with an aggregate access rate of 25 requests / sec. We ran four experiments. In the first one, we had no updates. In the second experiment, updates were made only to the 500 virt WebViews, at an aggregate rate of 5 updates / sec. In the third experiment, updates were made only to the 500 matweb WebViews, at a rate of 5 updates / sec. Finally, in the last experiment, both types of WebViews had updates, with an aggregate rate of 5 upd / sec.

Figure 9 depicts the results of our experiments. For each experiment, we report the average query response time of WebViews under the virt policy (left, light-colored column) and the average query response time for matweb WebViews (right, dark-colored column). As we showed in section 4.2, the average query response time for WebViews under the matweb policy changes very little with increases in the update workload, which agrees with the total cost formula and the results from this experiment. For virt WebViews however, there is a significant increase in the average query response time when there are updates, which also agrees with our formula. The case of updates on virt WebViews (second pair of columns) has 27% higher average query response times compared to the no updates case. When the updates are on matweb WebViews (third pair of columns) the increase in average query response time is even higher: 236% compared to the no updates case. The reason for this is that the updates on the matweb WebViews are using the DBMS, which has adverse effects on the performance of virt access queries. This was clearly predicted by the total cost formula, since we included the cost of updates on matweb WebViews in the case where there are other types of WebViews in the system (second line of Eq. 9, \(b = 1\)). The reason for such a big difference in our case is that except for putting more load on the DBMS, updates on matweb WebViews also compete against virt queries for resources inside the DBMS. In the case of virt updates, this did not happen, because both the queries and the updates were referring to the same tables.

5 Conclusions
WebView materialization can speed up the query response times of database-backed web servers significantly. However, the multi-tiered architecture of typical web servers and the need for online updates raise new issues, when compared to the view selection problem in data warehouses. In this paper, we compared three materialization policies: virtual (virt), materialized inside the DBMS (matdb) and materialized at the web server (matweb), both analytically and quantitatively. We developed a detailed cost model that takes
Acknowledgements

We would like to thank Manuel Rodriguez, Damianos Karakos and Yannis Kotidis for their suggestions, and the reviewers for their useful comments.

References


